JEM⋆: a different approach of heterogeneous parallel and distributed computing based on Java⋆⋆

Serge Chaumette and Asier Ugarte

LaBRI, Laboratoire Bordelais de Recherche en Informatique,
Université Bordeaux I, 351, Cours de la Libération,
33405 Talence, FRANCE

Serge.Chaumette@labri.u-bordeaux.fr
Asier.Ugarte@labri.u-bordeaux.fr

Abstract. JEM is a distributed platform based on Java-RMI/CORBA that makes it possible to access heterogeneous resources distributed over a network in a seamless manner. It provides the same interface to all resources integrated inside the system. In this paper we present two of the main features of the JEM platform. We first illustrate the usage of the desktop-like interface which we have developed on top of the kernel of the platform. This makes it possible for instance to drag a file over a printer to have it printed without knowing about physical locations. Second, we describe the units of execution which we have integrated inside the platform to handle fine-grain parallelism. These entities are some sort of mixture of mobile agents and remotely accessible threads. The possibility to make fine-grain parallelism provided by our platform is one of the features that make it different from existing systems. Indeed, existing systems provide a sort of worldwide virtual computer better suited for large-grain parallelism.

1 Introduction

Many research activities are being carried out around what is referred to as Meta, Web or Grid computing. The aim of the most significant projects is to provide the scientific computing community with a seamless virtual computer that would span the whole world. In the long term, such a system should provide a desktop-like user interface. This would make it possible to pilot the virtual computer using some sort of drag and drop technique. The aim of the platform which we are developing is to produce such a platform, still providing efficient fine-grain parallelism, which is often not in the scope of other platforms. First, it provides a Jini-like "plug-and-participate" structure [5]. This part of the platform

⋆ This work is partly supported by the Université Bordeaux I, the Region Aquitaine and the CNRS.

⋆⋆ Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. The authors are independent of Sun Microsystems, Inc.

will not be detailed here and the user is referred to [2] for more information. On top of this infrastructure we have developed a graphical desktop which is both a proof of concept and a mandatory tool if we want final users to use the system in a seamless manner. Second, it offers all the necessary infrastructure for a meta-computing platform, still providing the required mechanism that will make it possible to use fine-grain parallelism efficiently. In this paper we focus on two of the main features of JEM that make it different from other platforms. First, we describe its desktop like interface to network resources based on a volunteer plug and participate mechanism. Second, we present the units of execution of the platform, *Activities* used for mobility and *Tasks* used as remotly controllable threads. The platform is based on Java-RMI[1] and also runs on top of CORBA[7].

The rest of this paper is organized as follows. In section 2 we present projects which are in some manner related to what we are doing in the JEM platform. Section 3 shows the most significant features that make it possible to understand how the JEM platform works and how it is used through its desktop interface. We then concentrate on two of the major views an external user of the platform can have: a desktop distributed environment on one hand (section 3), and a possibly fine-grain parallel computing architecture (section 4) on the other hand. We eventually conclude in section 5 and sketch future directions of the project and usages of the JEM platform.

2 Related work

In this section we present the main features of those systems which JEM best compares to. These are Millenium, Jini, Legion and Globus. Many more systems exist, but they will not be presented here.

Millenium[8] is a Microsoft research project. The aim of this project is to provide a distributed system to "eliminate the distinction between distributed and local computing". To achieve this goal it provides "aggressive abstraction" that hides any physical information from the programmer. The system is self configuring: a new hardware equipment can join the system and can be removed in case of failure. Of course scalability, security and resource management are strong concerns in the project. Two prototypes have been developed, Coign and Continuum, that illustrate the operation of Millenium on COM components.

Jini[5] is a software architecture developed by Sun. Its aim is to provide a distributed "plug-and-participate" architecture. It makes it possible to share resources, to virtualize their location and to manage them easily. These resources can be physical, like printers for instance, or logical, like pieces of software. To join a Jini community, a component is plugged physically into the system. It joins the community by registering its service by a look up service. This service then becomes available to any other service willing to use it. To do so, a software component will query the lookup service. It will receive an interface to communicate with the service. There is then a direct communication using RMI with the service just located.

Globus[3] is a project, the aim of which is to provide the infrastructure to set-up a computational grid. These efforts come to life with the Globus Ubiquitous Supercomputing Testbed which is the interconnection of computers using the technologies and software components provided within the Globus "bag of services". The main basic principles can be summarized as an "hourglass" and translucent interfaces. The principle of the "hourglass"[3] is to provide a single simple interface to different available low-level services. Such an interface can be used to build new high level distributed services. For instance, the communication layer of Globus is the Nexus library. It can be implemented on top of several available low-level communication libraries (IP, shared memory, ..) and it can in turn be used to develop higher level libraries such as MPI. Nevertheless it might be worth controlling the way Nexus is mapped to the underlying protocol for instance to gain some efficiency when possible. This is what is referred to as translucent interfaces[3]: some attributes can be used to set-up parameters that control the mapping of Nexus to underlying layers.

Legion[4] is a project which is carried out at the University of Virginia. The aim of this project is to build a "Worldwide Virtual Computer". Legion is an object-based opened system. It supports scheduling, fault tolerance, site autonomy and many security possibilities. One of the main aims of the system is to provide users with parallelism that uses the underlying possibly many computers available worldwide. In legion everything is an object. Its architecture closely resembles the CORBA architecture. Objects in some sens publish methods that can be invoked by other objects. The naming mechanism of Legion closely resembles that of CORBA. Although Legion supports parallel libraries such as MPI, and wrapping of parallel components, it is not appropriate for fine-grain parallelism.

3 JEM as a desktop platform

JEM provides homogeneous access to the heterogeneous resources of a network. These resources can be physical – printers, screens, processors, ... – or logical – software code, C++ objects, ... –. This part of JEM resembles the Jini environment. Resources are wrapped inside some object glue that presents all of them with the same interface(figure 1). Each resource can then be fed using a `copyIn` method and results can be obtained using a `copyOut` method. The objects built from wrapping resources are then integrated in a tree-like structure (figures 1 and 3) similar to the Unix file system. We will not enter the details of the internal structure of the kernel in this paper. For further details the reader is referred to [2] that provides an in depth description of the kernel of JEM.

A desktop like interface has been developed. The aim of this "point and click" interface is to provide users with an environment which is straightforward to use. The fact that the components of the system are possibly distributed over a network is hidden from the final user. The user can easily manage all types of components simply using drag and drop. Figures 2 and 3 show this interface. For instance, the user can drag a file shown on the desktop to a printer. This causes the file to be printed. The user does not need to care about the physical locations of these objects in the network. This ability to migrate resources is made possible by the underlying platform.

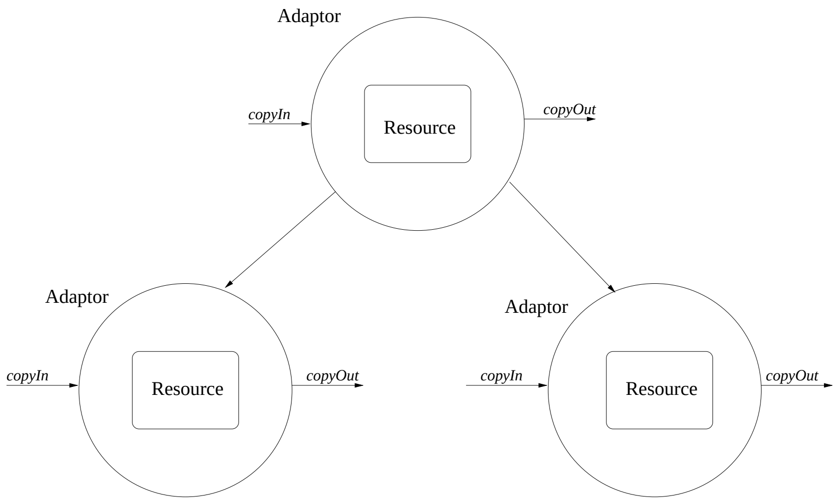


Fig. 1. Architecture of JEM

4 JEM as a parallel computing platform

4.1 Activities and tasks

It is quite naturally that JEM is also a distributed and parallel computing environment: processors are resources and can therefore be part of the platform.

We first considered threads as the basic execution unit. The problem with threads is that migrating them is not possible in standard Java Virtual Machines.

Therefore, we introduce two units of execution: *Activities* and *Tasks*. These provide the developer with powerfull and flexible mechanisms to implement distributed and parallel applications on the JEM platform. With these units of concurrency we support migration that will for instance be used for dynamic load balancing. We provide the programmers with an easy way to express stability (Activities) and unstability (Tasks). A program is stable if all its data can be saved and restored, i.e. it can be migrated.

Activities. An *activity* is some sort of mobile agent. It can migrate from host to host inside the JEM platform and resume its execution after migration. Activities are implemented keeping the JEM philosophy in mind: they are resources managed through the set of generic operations defined in the kernel (see section 3). They are handled the same way as a screen or a processor for instance, and are integrated in the hierarchical structure of the platform. Since we do not want to control the compilation, we are sometimes unable to prevent activities from doing operations that make them unstable for migration (for instance an

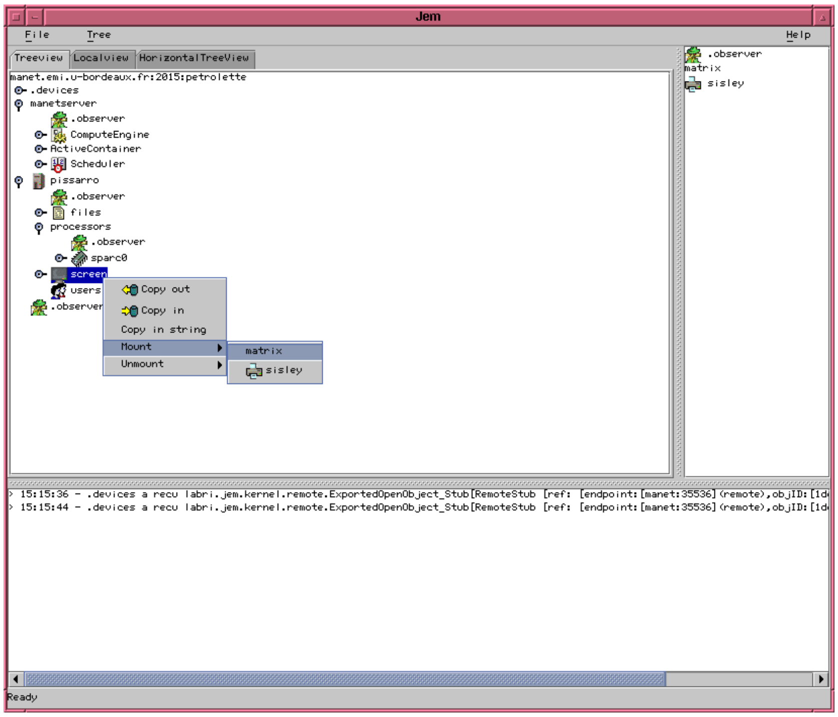


Fig. 2. The user desktop interface build on top of JEM

activity could access any physical local resource directly), therefore it is up to the programmer to ensure the stability of activities. When unstable code must be executed, the programmer has to use tasks.

Tasks. A *task* is an additional unit of execution used to insulate operations that imply unstability. The main difference between an activity and a task is that a task cannot migrate. It remains local to the host where it was started. Nevertheless, a task may be remote to the activity that created it, either because the activity migrated or because it was created remotely. It can be seen as a Java thread the execution of which may be initiated and controlled remotely.

4.2 How does it all work ?

In what follows, to talk about servers, we use the names of the machines where they are located. Servers are the objects that contain tasks and activities in the JEM hierarchy. Figure 4(a) shows an activity running inside the server Kediri. This activity remotely creates a task inside a server called Malang. The activity can communicate with the task through the `RunningTask` handle that

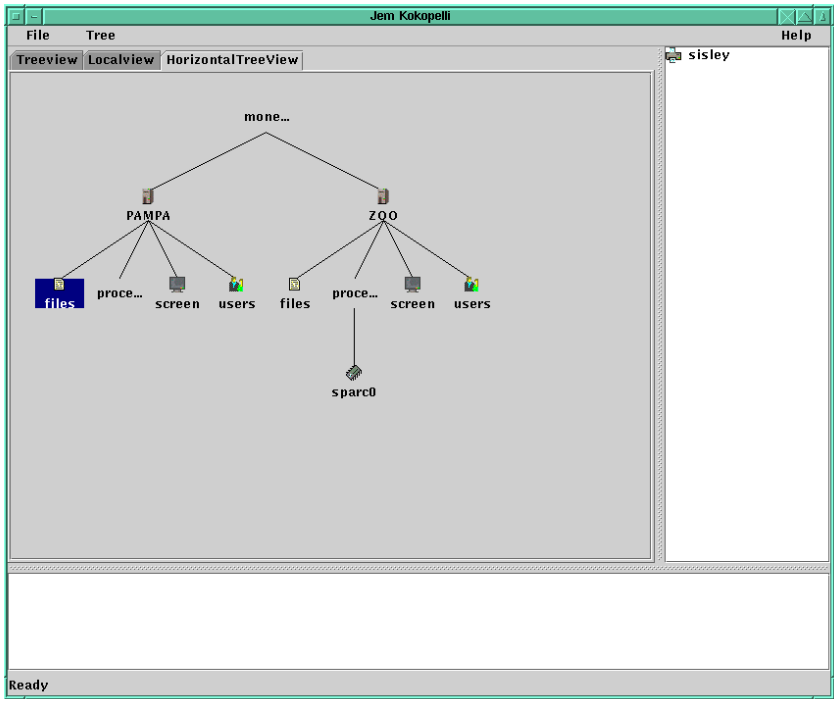


Fig. 3. The horizontal view mode

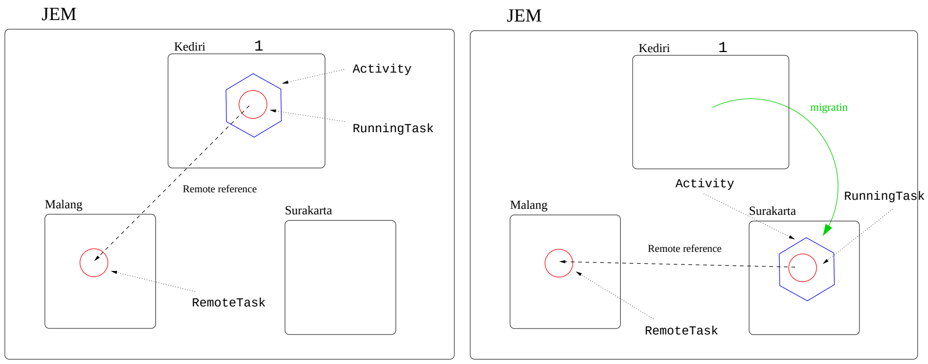
was returned by the system at starting time. The activity then migrates to the Surakarta server, as shown figure 4(b), carrying with it the `RunningTask` handle which enables it to communicate with the now remote task.

Activities For a programmer, the process of creating an activity consists in extending the `Activity` class shown program 1.

— Program 1 : Activity.java —

```
public class Activity implements java.io.Serializable,
                                 java.lang.Runnable{
  [...]
  public final void regulate(Location _location)[...]
  public void run(){}
  [...]
}
```

— Program 1 : Activity.java —



(a) Before migration (b) After migration

Fig. 4. Migration of an activity

The `Activity` class should not be directly instantiated. It must be extended to create activities inside the JEM platform and its `run` method has to be redefined. To migrate, an activity invokes the `regulate` method. This method can only be called by an activity on itself: it is up to the activity to decide when it can be migrated. The reason for this is that it is the only one to know when it is in a stable state that allows migration. Furthermore the choice of the policy to use to choose the destination of the migration is left to the activity (the policy can be based on the most powerfull server available, the less loaded, . . .).

— Program 2 : MyActivity.java —

```
public class MyActivity extends Activity{
  public void run(){
  [...]
  regulate(location_policy);
  [...]
  }
}
```

— Program 2 : MyActivity.java —

Tasks To create its own threads of execution, still keeping the capacity to migrate, an activity cannot use a `java.lang.Thread` because a thread cannot be migrated: it will use the `Task` class that we provide instead. `Task` is a `Runnable`, i.e. it implements this interface. To create a JEM task the programmer extends it and redefines its `run` method. When starting a task, the JEM kernel returns a handle to a `RunningTask`. A `RunningTask` is a proxy that allows a user to interact with a possibly remote task (see figure 4). An activity can migrate carrying a `RunningTask`, and continue to control the remote task by means of the `RunningTask` handle. The `RunningTask` is created by the JEM kernel when starting a task, and returned to the caller of the `start` method. The `Task` and the `RunningTask` classes should not be directly instantiated.

—— Program 3 : RunningTask.java ——

```
public final class RunningTask implements java.io.Serializable{

    [...]
    public String getName()[...]
    public int getPriority()[...]
    public void setPriority(int _priority)[...]
    public void join()[...]
    [...]
}
```

—— Program 3 : RunningTask.java ——

Note that a task, like an activity, is a JEM resource, and it respects all the principles defined in the JEM kernel[2]. Tasks make it possible for an activity to access local physical resources that otherwise would prevent the activity from migrating. In fact, the `RunningTask` will allow us to accomplish all the services usually found in a thread implementation : synchronizations, management of threads, scheduling, ... The details of the implementation will not be described here. Figures 4(a) and 4(b) illustrate the activity and task mechanisms.

—— Program 4 : MyApplication.java ——

```
public MyTask extends Task{
  public void run(){
    [...]
  }
}

public MyActivity(){
  public void run(){
     [...]
     MyTask _task = new MyTask();  // We create our own task
                                   //(a runnable object)

     RunningTask _runningTask      // The kernel starts the task
         = start(_task, _policy);  // in a server according to
                                   // the given policy
                                   // and returns a proxy

     [...]
```

```
     regulate(Location Policy);    // The activity asks to be
                                   // regulated (eventually migrated)
     [...]
     _runningTask.join();          // The activity synchronizes
                                   // with the started task
     [...]
   }
}

public class MyApplication implements Application{
  public void main(Context context, String args[])
         throws labri.jem.kernel.remote.TransportException{
    // This starts the activity in a JEM server
    MyActivity _activity = new MyActivity();
    [...]
  }
}
```

Program 4 : MyApplication.java

4.3 Task scheduling

JEM is a shared platform, thus the activities and tasks of different users might compete inside the same Java Virtual Machine. Therefore we provide a configurable scheduler. Figure 5 illustrates the operation of the currently available scheduler. It uses priority levels, and then round robin time sliced scheduling at a given priority level.

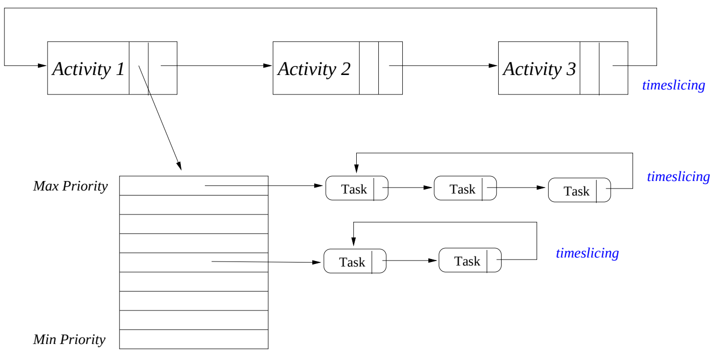


Fig. 5. The current scheduler policy

5 Conclusion and future work

JEM is still in the phase of development. Nevertheless, the development of the desktop, as a student project built on top of the kernel functionalities, has been successful. We keep on improving the functionalities, still getting better insight of what the kernel must be like, and how to make it even more opened so as to keep our developments adaptable. We are conscious that some mandatory features are still missing. Among these are access rights, encryption of communication, ... We believe that the new laws regarding the usage of encryption in France will make it easier for us to deal with these problems using public-key/private-key technics, credentials and access control lists. The implementation of security features will partly take place inside the kernel of the system and as dedicated objects in the JEM hierarchy.

The advantage of the JEM platform is that it can both provide large-grain and fine-grain parallelism. The next phase regarding this point is double. First we intend to use this feature to develop applications that would exploit our scheduling possibilities using fine-grain parallism. Second, we intend to take advantage of the migration possibilities of our platform even further. We are on the way to integrate the PM2[6] library in the JEM Platform. PM2 is a library that makes it possible to develop C threads that can migrate. Even though migrating in C is of course more limited than migrating in Java – where heterogeneous target platforms can be considered – we will most likely gain efficiency and be able to take legacy code into account.

As of writing, we are now in the process of considering our platform for deployment inside our Laboratory for system administration, and as a tool used within the framework of the parallel and distributed lecture of the Computer Science Engineer School which is an associated school of University Bordeaux 1.

References

1. K. Arnold and J. Gosling. *The Java programming language.* Addison-Wesley, 1996.
2. Serge Chaumette. JEM-DOOS: the Java based Distributed Objects Operating System of the JEM project. In Denis Caromel and al., editors, *Proceedings of the Second International Symposium on Object Oriented Parallel Environments, ISCOPE 98, Santa Fe, NM, USA*, volume 1505, pages 135–142, december 1998.
3. Ian Foster and Kesselman Car. The globus project: A status report. In *Proceedings of IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.
4. A.S. Grimshaw, A. Wulf, and the Legion team. Legion. The next logical step toward the world-wide virtual computer. *Communications of the ACM*, 40(1), 1997.
5. Sun microsystems. Jini architectural overview, a technical white paper. Technical report, 1999.
6. Jean-François Méhaut Raymond NAMYST. pm^2 : Parallel multithreaded machine, a computing environment for distributed architectures. Technical report, LIFL, Villeneuve d'Ascq Cedex. Available at URL http://www.lifl.fr/ namyst/ps/survey.ps.
7. J. Siegel. *CORBA, Fundamental and Programming.* Wiley, 1996.
8. Robert P. Fitzgerald Christopher W. Fraser Michael B. Jones Todd B. Knoblock Rick Rashid William J. Bolosky, Richard P. Draves. Operating system directions for the next millenium.